A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations

Filip Pawłowski[a,b,*], Bora Uçar[c], Albert-Jan Yzelman[a]

[a] *Huawei Technologies France*
20 Quai du Point du Jour, 92100 Boulogne-Billancourt, France
[b] *ENS Lyon, France*
[c] *Univ Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, LIP UMR 5668, F-69007 Lyon, France*

Abstract

Computation on tensors, treated as multidimensional arrays, revolve around generalized basic linear algebra subroutines (BLAS). We propose a novel data structure in which tensors are blocked and blocks are stored in an order determined by Morton order. This is not only proposed for efficiency reasons, but also to induce efficient performance regardless of which mode a generalized BLAS call is invoked for; we coin the term *mode-oblivious* to describe data structures and algorithms that induce such behavior. Experiments on one of the most bandwidth-bound generalized BLAS kernel, the tensor–vector multiplication, not only demonstrate superior performance over two state-of-the-art variants by up to 18%, but additionally show that the proposed data structure induces a 71% less sample standard deviation for tensor–vector multiplication across d modes, where d varies from 2 to 10. Finally, we show our data structure naturally expands to other tensor kernels and demonstrate up to 38% higher performance for the higher-order power method.

Keywords: tensor computations, data structure, Morton order, tensor–vector multiplication

2010 MSC: 15A72, 68P05

*Filip Pawłowski, Tel: +0033763251489; filip.pawlowski1@huawei.com
Email addresses: filip.pawlowski1@huawei.com (Filip Pawłowski), bora.ucar@ens-lyon.fr (Bora Uçar), albertjan.yzelman@huawei.com (Albert-Jan Yzelman)

1. Introduction

We investigate computations on dense tensors (or multidimensional arrays) in d modes (dimensions). Tensor operations apply to specific modes; tensor–vector multiplication (TVM), tensor–matrix multiplication (TMM), and tensor–tensor multiplication (TTM), e.g., each operate on a subset of modes of the input tensor. We dub algorithms such as the TVM, TMM, and TTM *kernels*. Much in line with the original BLAS definition [7, 8], we classify the TVM as a generalized BLAS level-2 (BLAS2) kernel, while we classify the TMM, TTM, and Khatri-Rao products [10] as generalized BLAS3 ones. These kernels form core components in tensor computation algorithms [1]; one example is the computation of Candecomp/Parafac decomposition of tensors using the alternating least squares method [2] and its computationally efficient implementations [10, 21, 13].

We define a tensor kernel to be *mode-aware* if its performance strongly depends on the mode in which the kernel is applied; otherwise, we define the kernel to be *mode-oblivious*. This informal definition is in-line with the more widely known concept of cache-aware versus cache-oblivious algorithms [9]. We propose blockwise storage for tensors to mode-obliviously support common tensor kernels. We closely investigate the TVM kernel, which is the most bandwidth-bound and thus the most difficult one to achieve high performance for. Efficient TMM and TTM kernels, in contrast, often make use of the compute-bound general matrix–matrix multiplication (BLAS3).

Tensors are commonly stored in an *unfolded* fashion, which corresponds to a higher-dimensional equivalent to row-major or column-major storage for matrices; while a matrix can be unfolded in two different ways, d-dimensional tensors can be stored in $d!$ different ways. Section 2 discusses previous work in tensor computations, including tensor storage. Section 3 continues by first developing a notation for precisely describing a tensor layout in computer memory and for describing how an algorithm operates on tensor data stored as such.

Section 4 discusses various ways for implementing the TVM. It first notes

its similarity to the matrix–vector multiplication (MVM): it takes a tensor, the index of a mode, and a vector of size conformal to that mode's size and performs scalar multiply and add operations. In fact the MVM kernel can be used to carry out a TVM by either i) reorganizing the tensor in memory (*unfolding* the tensor) followed by a single MVM, or ii) reinterpreting the tensor as a series of matrices, on which a series of MVM operations are executed. We describe how to implement them using BLAS2, resulting in two highly optimized baseline methods. The section then introduces our proposed blocked data structure for efficient, mode-oblivious performance. A blocked tensor is a tensor with smaller equally-sized tensors as its elements. We consider only the case where smaller tensor blocks are stored in an unfolded fashion and are processed using one or more BLAS2 calls. We define two *block layouts*, which determine the order of processing of the smaller blocks: either a simple ordering of dimensions or one inferred from the Morton order [20].

The experiments in Section 5 show that the Morton order blocked data structure offers higher performance on the TVM kernel when compared to the state-of-the-art methods, while maintaining a significantly lower standard deviation of performance when applied on the various modes, thus indeed achieving mode-oblivious behavior. We show the proposed data structure easily extends to other tensor kernels by implementing a higher-order power method, and show the superior performance observed for the TVM is retained. We conclude and suggest future work in Section 6.

2. Related work

To the best of our knowledge, ours is the first work discussing a blocking approach for obtaining efficient, mode-oblivious tensor computations. A dense tensor–vector multiplication routine is closely related to the BLAS2, of which there are many implementations: some well-known are OpenBLAS [28], ATLAS [27], and Intel MKL [12]. BLIS is a code generator library that can emit BLAS kernels which operate without the need to reorganize input matrices in

case elements are strided [25]. However, strided algorithms tend to perform worse than direct BLAS calls when those can be made instead; which for most tensor computations is possible [16].

Li et al. [16] discuss an algorithm for the *TMM* that uses BLAS3 routines and an auto-tuning approach. They compare their work to an unfolding-based approach, common in the related literature [15]. This latter approach explicitly unfolds the tensor storage to use an optimized matrix–matrix (*MM*) multiplication kernel directly. The unfolding-based approach not only requires unfolding of the input, but also requires unfolding of the output. Li et al. instead propose a parallel loop-based algorithm: a loop of the BLAS3 kernels which operate in-place on parts of the tensor such that no unfolding is required. They use some heuristics and two microbenchmarks to decide on the size of the *MM* kernel and the distribution of the threads among the loops. Li et al. do not discuss blocking explicitly.

Kjolstad et al. [14] propose The Tensor Algebra Compiler (taco) for tensor computations. It generates code, mixing dense and sparse storages for different modes of a tensor according to the operands of a tensor algebraic expression. Supported formats aside from the dense unfolded storage are Compressed Sparse Row (otherwise known as Compressed Row Storage, CSR/CRS), its column-oriented variant, and (by recursive use of CSR/CSC) Compressed Sparse Fibers [22]. Lorton and Wise [18] use the Morton order within a blocked data structure for dense matrices, for the matrix–matrix multiplication operation. Yzelman and Bisseling [29] discuss the use of the Hilbert space-filling curve for the sparse matrix–vector multiplication. Both studies are motivated by cache-obliviousness and did not consider mode-obliviousness. Walker [26] investigates Morton ordering for 2D arrays to obtain efficient memory access in parallel systems. In recent work [17], Li et al. propose a data structure for sparse tensors which uses the Morton order to sort individual nonzero elements of a sparse tensor to put them in blocks, for efficient representation of sparse tensors.

A related and more computationally involved operation, known as "tensor–

tensor contraction" have received considerable attention. This operation is the most general form of the multiplication operation in (multi)linear algebra. CTF [23], TBLIS [19], and GETT [24] are recent libraries carrying out this operation based on principles and lessons learned for high performance matrix–matrix multiplication. Apart from not explicitly considering TVM, these do not adapt the tensor layout. As a consequence, they all require transpositions.

3. Background and notation

We use the notation taken in part from Kolda and Bader [15]. We use a calligraphic font for tensors, e.g., $\mathcal{A}$, capital letters for matrices and regular letters for vectors. We refer to individual dimensions using integers from 0 to $d-1$ and use $n_0 \times n_1 \times \cdots \times n_{d-1}$ to denote the sizes of a d-dimensional tensor $\mathcal{A}$; such a tensor has $\Pi_{i=0}^{d-1} n_i$ elements. We assume tensors have real values; although the discussion can apply to other fields.

The k-mode tensor–vector multiplication of a tensor $\mathcal{A} \in \mathbb{R}^{n_0 \times n_1 \times \cdots \times n_{d-1}}$ with a vector $v \in \mathbb{R}^{n_k}$ is denoted by the symbol $\overline{\times_k}$:

$$\mathcal{P} = \mathcal{A} \ \overline{\times_k} \ v \quad \text{where} \quad \mathcal{P} \in \mathbb{R}^{n_0 \times n_1 \times \ldots n_{k-1} \times 1 \times n_{k+1} \ldots \times n_{d-1}} ,$$

where for all $i_0, i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_{d-1}$,

$$\mathcal{P}_{i_0,i_1,\ldots,i_{k-1},0,i_{k+1},\ldots,i_{d-1}} = \sum_{i_k=0}^{n_k-1} \mathcal{A}_{i_0,i_1,\ldots,i_{d-1}} v_{i_k} .$$

The above formulation is a contraction of the tensor along the kth mode. We assume that the operation does not drop the contracted mode; the resulting tensor is always d-dimensional, where the kth dimension is of size one. For the advantages of this formulation see Bader and Kolda [1, Section 3.2].

In this paper we shall use a flat notation. For example, the following represents an order-3 tensor using a visual separation between the slices 0 (lower left

quadrant) and 1 (top right quadrant):

$$\mathcal{B} = \begin{pmatrix} & & & & 41 & 43 & 47 & 53 \\ & & & & 59 & 61 & 67 & 71 \\ & & & & 73 & 79 & 83 & 89 \\ 2 & 3 & 5 & 7 & & & & \\ 11 & 13 & 17 & 19 & & & & \\ 23 & 29 & 31 & 37 & & & & \end{pmatrix} \in \mathbb{R}^{3\times 4\times 2}\,. \tag{3.1}$$

A **layout** of a tensor $\mathcal{A}$, denoted as $\rho(\mathcal{A})$, is a function which maps tensor elements $\mathcal{A}_{i_0,i_1,\ldots,i_{d-1}}$ onto an array of size $\Pi_{i=0}^{d-1} n_i$:

$$\rho(\mathcal{A}) : \{0,1,\ldots,n_0-1\} \times \{0,1,\ldots,n_1-1\} \times \cdots \times \{0,1,\ldots,n_{d-1}-1\} \mapsto \{0,1,\ldots,\Pi_{i=0}^{d-1} n_i - 1\}.$$

A layout of a tensor defines the order in which the tensor elements are stored in computer memory. We always assume that a tensor is stored in a contiguous memory area.

Let $\rho_\pi(\mathcal{A})$ be a layout associated with a permutation π of $(0,1,\ldots,d-1)$ such that

$$\rho_\pi(\mathcal{A}) : (i_0,i_1,\ldots,i_{d-1}) \mapsto \sum_{k=0}^{d-1} \left(i_{\pi_k} \prod_{j=k+1}^{d-1} n_{\pi_j} \right)\,. \tag{3.2}$$

Conversely, the ith element in memory corresponds to the tensor element with coordinates given by the inverse of the layout $\rho_\pi^{-1}(\mathcal{A})$:

$$i_k = \left\lfloor \frac{i}{\Pi_{j=k+1}^{d-1} n_{\pi_j}} \right\rfloor \mod n_k, \text{ for all } k \in \{0,\ldots,d-1\}\,.$$

For matrices, this relates to the concept of row-major and column-major layout, which, using the layout definition (3.2), correspond to $\rho_{(0,1)}(A)$ and $\rho_{(1,0)}(A)$, respectively. Such a permutation-based layout is called a tensor **unfolding** [15] and describes the case where a tensor is stored as a regular multidimensional array.

Let $\rho_Z(\mathcal{A})$ be a **Morton layout** defined by the space-filling Morton order [20]. The Morton order is defined recursively, where at every step the covered space is subdivided into two within every dimension; for 2D planar areas

this creates four cells, while for 3D it creates eight cells. In every two dimensions the order between cells is given by a (possibly rotated) Z. Let w be the number of bits used to represent a single coordinate, and let $i_k = (l_0^k l_1^k \dots l_{w-1}^k)_2$ for $k = 0, 1, \dots, d-1$ be the bit representation of each coordinate. The Morton order in d dimensions $\rho_Z(\mathcal{A})$ can then be defined as

$$\rho_Z(\mathcal{A}) : (i_0, i_1, \dots, i_{d-1}) \mapsto (l_0^0 l_0^1 \dots l_0^{d-1} l_1^0 l_1^1 \dots l_1^{d-1} \dots l_{w-1}^0 l_{w-1}^1 \dots l_{w-1}^{d-1})_2 \,. \tag{3.3}$$

The inverse $\rho_Z^{-1}(\mathcal{A})$ yields the coordinates of the ith consecutively stored element in memory, where $i = (l_0 l_1 \dots l_{d^w-1})_2$:

$$\rho_Z^{-1}(\mathcal{A}) : i \mapsto (i_0, i_1, \dots, i_{d-1}) \quad \text{where} \quad i_k = (l_{k+0d} l_{k+1d} \dots l_{k+(w-1)d})_2 \,, \tag{3.4}$$

for all $k \in \{0, 1, \dots, d-1\}$.

Let $M_{\rho_\pi}^{k \times l}(\mathcal{A})$ be the **matricization** of $\mathcal{A}$ which views a tensor layout $\rho_\pi(\mathcal{A})$ as a $k \times l$ matrix:

$$M_{\rho_\pi}^{k \times l}(\mathcal{A}) : \mathbb{R}^{n_0 \times n_1 \times \dots \times n_{d-1}} \mapsto \mathbb{R}^{k \times l} \,,$$

where π is a permutation of $(0, \dots, d-1)$, $kl = \Pi_{i=0}^{d-1} n_i$, and $k = \Pi_{k=0}^{b} n_{\pi_k}$ for some $0 \leqslant b \leqslant d-1$. We relate the entries $(M_{\rho_\pi}^{k \times l}(\mathcal{A}))_{i,j}$ to $\mathcal{A}_{i_0, i_1, \dots, i_{d-1}}$ by

$$i = \sum_{k=0}^{b} i_{\pi_k} \prod_{j=k+1}^{b} n_{\pi_j} \quad \text{and} \quad j = \sum_{k=b+1}^{d-1} i_{\pi_k} \prod_{j=k+1}^{d-1} n_{\pi_j} \,,$$

where $i \in \{0, 1, \dots, k-1\}$, $j \in \{0, 1, \dots, l-1\}$ and $i_k \in \{0, 1, \dots, n_k - 1\}$. For example, $M_{\rho_{\boldsymbol{\pi}}}^{3 \times 8}(\mathcal{B})$ corresponds to the following $n_0 \times n_1 n_2$ matricization of $\rho_{(0,1,2)}(\mathcal{B})$ (3.1):

$$M_{\rho_{\boldsymbol{\pi}}}^{3 \times 8}(\mathcal{B}) = \begin{pmatrix} 2 & 41 & 3 & 43 & 5 & 47 & 7 & 53 \\ 11 & 59 & 13 & 61 & 17 & 67 & 19 & 71 \\ 23 & 73 & 29 & 79 & 31 & 83 & 37 & 89 \end{pmatrix} \in \mathbb{R}^{3 \times 8} \,.$$

4. Algorithms for tensor–vector multiplication

In this section, we describe two algorithms which correspond to the current state of the art. Additionally, we propose two block algorithms to perform

tensor–vector multiplication. For the state-of-the-art TVM algorithms, the input tensor is stored using a ρ_π layout. For the block TVM algorithms, we discuss blocked layouts in Subsection 4.2. For performance we do not modify the data structure while performing a TVM, and retain the permutation π for the output tensor.

The number of floating point operations (flops) of a k-mode TVM is $2\Pi_{i=0}^{d-1} n_i$. The minimum number of data elements touched is:

$$\Pi_{i=0}^{d-1} n_i + \frac{\Pi_{i=0}^{d-1} n_i}{n_k} + n_k \ , \tag{4.1}$$

where $\Pi_{i=0}^{d-1} n_i$ is the size of the input tensor, $\frac{\Pi_{i=0}^{d-1} n_i}{n_k}$ is the size of the output tensor, and n_k is the size of the input vector. The arithmetic intensity of the TVM is the ratio of its floating point operations to its memory accesses:

$$\frac{2\Pi_{i=0}^{d-1} n_i}{w(\Pi_{i=0}^{d-1} n_i + \frac{\Pi_{i=0}^{d-1} n_i}{n_k} + n_k)} \ \textit{flop per byte} \, , \tag{4.2}$$

where w is the number of bytes required to store a single element. This lies between $1/w$ and $2/w$ and thus represents an extremely bandwidth-bound operation. The MVM is in same range of arithmetic intensity.

We cast all TVM algorithms formulated in this section as one or more calls to one of two types of $MVMs$: the left-hand sided multiplication vm ($u = vA$) and the right-hand sided multiplication mv ($u = Av$). Both assume a $\rho_{(0,1)}$ layout for A; i.e., a row-major storage. Both MVM operations correspond to standard BLAS2 calls, thus enabling the use of state-of-the-art BLAS libraries.

4.1. Two state-of-the-art tensor–vector multiplication algorithms

First, we discuss two common algorithms to compute a k-mode TVM assuming a tensor with ρ_π layout. Algorithm 1 repeatedly invokes a column-major MVM on consecutive parts of the tensor in-place, by matricization. Algorithm 2 instead reorders the tensor in memory such that the data is aligned for a single column-major MVM.

Both algorithms rely on a single MVM kernel for the case when $\pi_0 = k$ or $\pi_{d-1} = k$, in which case the memory touched by the MVM kernel corresponds

Algorithm 1 *tvLooped*$(\mathcal{A}, v, k, \rho_\pi(\mathcal{A}))$: the looped tensor–vector multiplication

Input: An $n_0 \times n_1 \times \cdots \times n_{d-1}$ tensor $\mathcal{A}$ with $\rho_\pi(\mathcal{A})$
An $n_k \times 1$ vector v, for $k \in \{0, 1, \ldots, d-1\}$
Output: An $n_0 \times n_1 \times \cdots \times n_{k-1} \times 1 \times n_{k+1} \times \cdots \times n_{d-1}$ tensor $\mathcal{B}$,
$\mathcal{B} = \mathcal{A}\overline{\times_k}v$ with $\rho_\pi(\mathcal{B})$

$n = \Pi_{i=0}^{d-1} n_i$ ▶ Number of tensor elements.
if π_{d-1} **equals** k **then**
  Let $A = M_{\rho_\pi}^{n/n_k \times n_k}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as a tall-skinny $\rho_{(0,1)}$-matrix.
  $u \leftarrow mv(A, v)$ ▶ A single mv computes $\mathcal{B}$.
  return $\mathcal{B} = (M_{\rho_\pi}^{n/n_k \times 1})^{-1}(u)$ ▶ Reinterpret u as a tensor.
else
  Let $r = \prod_{i=\pi_k^{-1}}^{d-1} n_{\pi_i}$ and $s = r/n_k$
  Let $A = M_{\rho_\pi}^{(n/r)n_k \times s}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as n/r wide $\rho_{(0,1)}$-matrices.
  Let B be an $n/r \times s$ matrix with $\rho_{(0,1)}(B)$ ▶ n/r vectors of length s.
  for $i = 0$ to $(n/r) - 1$ **do**
    $B_{i,:} \leftarrow vm(v^T, A_{in_k:(i+1)n_k,:})$
  return $\mathcal{B} = (M_{\rho_\pi}^{n/r \times s})^{-1}(B)$ ▶ Reinterpret B as a tensor.

Algorithm 2 *tvUnfold*$(\mathcal{A}, v, k, \rho_\pi(\mathcal{A}))$: the unfold tensor–vector multiplication

Input: An $n_0 \times n_1 \times \cdots \times n_{d-1}$ tensor $\mathcal{A}$ with $\rho_\pi(\mathcal{A})$
An $n_k \times 1$ vector v, for $k \in \{0, 1, \ldots, d-1\}$
Output: An $n_0 \times n_1 \times \cdots \times n_{k-1} \times 1 \times n_{k+1} \times \cdots \times n_{d-1}$ tensor $\mathcal{B}$,
$\mathcal{B} = \mathcal{A}\overline{\times_k}v$ with $\rho_\pi(\mathcal{B})$

$n = \Pi_{i=0}^{d-1} n_i$ ▶ Number of tensor elements.
if π_{d-1} **equals** k **then**
  Let $A = M_{\rho_\pi}^{n/n_k \times n_k}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as a tall-skinny $\rho_{(0,1)}$-matrix.
  $u \leftarrow mv(A, v)$
  return $\mathcal{B} = (M_{\rho_\pi}^{n/n_k \times 1})^{-1}(u)$
else if π_0 **equals** k **then**
  Let $A = M_{\rho_\pi}^{n_k \times n/n_k}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as a wide $\rho_{(0,1)}$-matrix.
  $u \leftarrow vm(v^T, A)$
  return $\mathcal{B} = (M_{\rho_\pi}^{1 \times n/n_k})^{-1}(u)$
else
  Let $r = \prod_{i=\pi_k^{-1}}^{d-1} n_{\pi_i}$ and $s = r/n_k$
  Let $A = M_{\rho_\pi}^{(n/r)n_k \times s}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as n/r wide matrices.
  Let U be an empty $n_k \times n/n_k$ matrix with layout $\rho_{(0,1)}(U)$
  for $i = 0$ to $(n/r) - 1$ **do**
    for $j = 0$ to $n_k - 1$ **do**
      $U_{j,is:(i+1)s} = A_{in_k+j,:}$ ▶ Rearrange A into U (tensor unfolding).
  $u \leftarrow vm(v^T, U)$ ▶ A single vm can now compute $\mathcal{B}$.
  return $\mathcal{B} = (M_{\rho_\pi}^{1 \times n/n_k})^{-1}(u)$ ▶ Reinterpret u as a tensor.

to the minimum number of elements touched (4.1). For all other modes, the two algorithms exhibit different behavior. Algorithm 1 touches at least

$$\prod_{i=\pi_k^{-1}}^{d-1} n_{\pi_i} + \prod_{i=\pi_k^{-1}+1}^{d-1} n_{\pi_i} + n_k \tag{4.3}$$

data elements for each of the $\prod_{i=0}^{\pi_k^{-1}-1} n_{\pi_i}$ MVM calls. This brings the data movement overhead of Algorithm 1 to

$$\left(\left[\prod_{i=0}^{\pi_k^{-1}-1} n_{\pi_i} \right] - 1 \right) n_k. \tag{4.4}$$

Algorithm 2 performs an explicit unfold of the tensor memory which instead incurs an overhead of $2\Pi_{i=0}^{d-1} n_i$. We choose a $\rho_{(1,0)}$-layout for the unfolded U instead of a $\rho_{(0,1)}$-layout since the latter would require element-by-element copies, while the former can copy ranges of size $\prod_{i=\pi_k^{-1}+1}^{d-1} n_{\pi_i}$. Furthermore, the former accesses the input tensor consecutively while individual accesses on the unfold matrix are interleaved; this is faster than the reverse.

4.2. Block tensor–vector multiplication algorithms

An order-d blocked tensor $\mathcal{A} \in \mathbb{R}^{n_0 \times n_1 \times \cdots \times n_{d-1}}$ consists of a total of $\prod_{i=0}^{d-1} a_i$ blocks $\mathcal{A}_j \in \mathbb{R}^{b_0 \times b_1 \times \cdots \times b_{d-1}}$ where $n_k = a_k b_k$ for all $k \in \{0, 1, \ldots, d-1\}$. A **blocked layout** for blocked tensors stores each block consecutively in memory. Individual blocks use a uniform layout. Given a blocked layout $\rho_0\rho_1(\mathcal{A})$, a block is stored as the $\rho_0(\mathcal{A})(i_0, i_1, \ldots, i_{d-1})$th block in the memory occupied by the tensor, while an element of a block is stored as the $\rho_1(A_0)(i_0, i_1, \ldots, i_{d-1})$th in the memory occupied by the block. It is thus a combination of two layouts: ρ_0 at the block-level, and ρ_1 within blocks.

We store blocks with ρ_π layout to take advantage of the *TVM* algorithms from Section 4.1 that exploit highly optimized BLAS2 routines. Algorithm 3 is a general block *TVM* algorithm which visits the blocks in the order imposed by the ρ_0 layout. When the *TVM* of a block finishes, the next block offset in the output tensor and the associated positions of the vector entry are computed using the *nextBlock* function, which implements the block order.

Algorithm 3 $btv(\mathcal{A}, v, k, nextBlock_{\rho_0}, tv)$: the block tensor–vector multiplication algorithm

Input: An $n_0 \times n_1 \times \cdots \times n_{d-1}$ blocked tensor $\mathcal{A}$ with $\rho_0\rho_\pi(\mathcal{A})$ consisting of $\Pi_{i=0}^{d-1} a_i$ blocks $\mathcal{A}_j \in \mathbb{R}^{b_0 \times b_1 \times \ldots \times b_{d-1}}$
An $n_k \times 1$ vector v, for $k \in \{0, 1, \ldots, d-1\}$
A $nextBlock_{\rho_0}$ function for indices of result o and vector i_k
A *TVM* algorithm tv for ρ_π layouts

Output: An $n_0 \times \cdots \times n_{k-1} \times 1 \times n_{k+1} \times \cdots \times n_{d-1}$ blocked tensor $\mathcal{B}$ consisting of $\Pi_{i=0}^{d-1} a_i / a_k$ blocks $\mathcal{B}_k \in \mathbb{R}^{b_0 \times \cdots \times b_{k-1} \times 1 \times b_{k+1} \times \cdots \times b_{d-1}}$, $\mathcal{B} = \mathcal{A} \overline{\times_k} v$ with $\rho_0\rho_\pi(\mathcal{B})$

1: Let $\mathcal{B}$ be a blocked tensor with layout $\rho_0\rho_\pi(\mathcal{B})$ with entries initialized to 0.
2: $(i_0, i_1, \ldots, i_{d-1}) \leftarrow \rho_0^{-1}(\mathcal{A})(0)$ ▶ Get coordinates of the first block.
3: $o \leftarrow \rho_0(\mathcal{B})(i_0, \ldots i_{k-1}, 0, i_{k+1}, \ldots, i_{d-1})$ ▶ Get output block index.
4: **for** $i = 0$ to $\Pi_{j=0}^{d-1} a_j - 1$ **do**
5: $\quad \mathcal{B}_o \leftarrow \mathcal{B}_o + tv(\mathcal{A}_i, v_{(i_k)b_k:(i_k+1)b_k}, k, \rho_\pi(\mathcal{A}_i))$
6: $\quad (o, i_k) \leftarrow nextBlock_{\rho_0}(k, \rho_0(\mathcal{A}), \rho_0(\mathcal{B}), i, o, i_k)$
7: **return** $\mathcal{B}$

We propose two blocked layouts: (i) $\rho_\pi\rho_\pi$, where the blocks are ordered using a permutation of dimensions; and (ii) $\rho_Z\rho_\pi$, where blocks are ordered according to the Morton layout. Depending on the blocked layout, the *nextBlock* function in Algorithm 3 then corresponds to $nextBlock_{\rho_\pi}$ (Algorithm 4) or $nextBlock_{\rho_Z}$ (Algorithm 5). The $nextBlock_{\rho_\pi}$ function has an efficient $\Theta(1)$ implementation which avoids explicitly evaluating ρ_π and ρ_π^{-1}. The $nextBlock_{\rho_Z}$ function calculates the indices by evaluating the Morton layout ρ_Z (3.3) and its inverse (3.4), which both cost $\Theta(d)$ time, space, and memory movement.

We do not explicitly evaluate ρ_Z^{-1}, but instead compute it incrementally while progressing from one block to the next, whose amortized analysis [4, ch. 17] yields a run time complexity of $\Theta(\Pi_{i=0}^{d-1} a_i)$ over the whole $\rho_Z\rho_\pi$-btv computation. Similarly, the evaluation of ρ_Z can be amortized and result in $\Theta(a)$ overall runtime cost. The memory overhead of ρ_Z using our efficient implementation is $\Theta(d + \log_2 \max_i a_i)$, due to maintaining a counter for each dimension, and for each level of the Morton order.

Since even a $\Theta(d\Pi_{i=0}^{d-1} a_i)$ overhead is much smaller than the number of operations the bandwidth-bound *TVM* performs, we expect neither the $\rho_\pi\rho_\pi$- nor the $\rho_Z\rho_\pi$-block *TVM* to slow down for this reason while we do expect significant

Algorithm 4 *nextBlock*$_{\rho_\pi}(k, \rho_\pi(\mathcal{A}), \rho_\pi(\mathcal{B}), i, o, i_k)$: the next block according to a ρ_π layout

Input: A mode k, and current indices of block i, result o, and vector i_k
The $\rho_\pi(\mathcal{A})$ input and $\rho_\pi(\mathcal{B})$ output tensor layouts are not used
Output: A result index o and a vector index i_k for the next block

Let $m_{right} = \prod_{i=\pi_k^{-1}}^{d-1} a_{\pi_i}$ and $m_{mode} = m_{right}/a_k$
$i \leftarrow i + 1$
if $(k > 0)$ **and** $((i \bmod m_{right})$ **equals** $0)$ **then**
    $o \leftarrow o + 1$
    $i_k \leftarrow 0$
else if $((i \bmod m_{mode})$ **equals** $0)$ **then**
    $o \leftarrow o - m_{mode} + 1$
    $i_k \leftarrow i_k + 1$
else
    $o \leftarrow o + 1$
return (o, i_k)

Algorithm 5 *nextBlock*$_{\rho_Z}(k, \rho_Z(\mathcal{A}), \rho_Z(\mathcal{B}), i, o, i_k)$: the next block according to a Morton layout

Input: A current block index i
The $\rho_Z(\mathcal{A})$ input and $\rho_Z(\mathcal{B})$ output tensor layouts
A mode k, and indices of output o and vector i_k are not used
Output: A result index o and a vector index i_k for the next block

$(i_0, \ldots, i_{d-1}) \leftarrow \rho_Z^{-1}(\mathcal{A})(i + 1)$ ▶ Get coordinates of the next block.
$o \leftarrow \rho_Z(\mathcal{B})(i_0, \ldots i_{k-1}, 0, i_{k+1}, \ldots, i_{d-1})$ ▶ Get output block index.
return (o, i_k)

and mode-oblivious increases due to cache reuse.

5. Experiments

We evaluate the proposed blocked tensor layouts for the *TVM* computation, evaluate their mode-obliviousness, and compare against the state of the art. Section 5.1 first presents our experimental setup and methodologies. To ascertain practical upper bounds for the performance of a *TVM*, Section 5.2 presents microbenchmarks designed to find realistic bounds on data movement and computation. We then follow with the assessment of the state-of-the-art *TVM* algorithms in Section 5.3 and the block *TVM* algorithms in Section 5.4 and compare them with the codes generated by the Tensor Algebra Compiler (taco) in Section 5.5. To show our proposed tensor layouts transfer to other tensor kernels as well as to other systems, we apply them to the iterative higher-order power method (HOPM) [5, 6] in Section 5.6.

5.1. Setup

We run our experiments on a single Intel Ivy Bridge node, containing two Intel Xeon E5-2690 v2 processors that are each equipped with 10 cores. The cores run at 3.0 GHz with AVX capabilities, amounting to 240 Gflop/s per processor. Each processor has 32 KB of L1 cache memory per core, 256 KB of L2 cache memory per core, and 25 MB of L3 cache memory shared amongst the cores. Each processor has 128 GB of local memory configured in quad-channel at 1600 MHz, yielding a theoretical bandwidth of 47.68 GB/s per socket. The system uses CentOS 7 with Linux kernel 3.10.0 and software is compiled using GCC version 6.1. We use Intel MKL version 2018.1.199 and LIBXSMM version 1.9-864.

Benchmarking methodology

We benchmark tensors of order-two ($d = 2$) up to order-10 ($d = 10$) and for simplicity assume square tensors of size n. We assume users are interested in input tensors that do not fit into cache, and thus choose n such that the combined input and output memory areas during a single *TVM* call have a

combined size of at least several GBs to make sure we capture out-of-cache behavior.

To benchmark a kernel, we first time a single run and calculate the number m of calls required to reach at least one second of run time. We then conduct 10 experiments as follows: we i) issue a sleep command for 1 second, ii) run the kernel once without timing, iii) time m runs of the kernel, and iv) store the time taken divided by m as t_i. Based on 10 experiments, we compute the average time $t_{\text{avg}} = (\sum_{i=0}^{m-1} t_i)/10$ and the (unbiased) sample standard deviation $t_{\text{std}} = \sqrt{\frac{1}{9}\sum_{i=0}^{9}(t_i - t_{\text{avg}})^2}$. Throughout experiments, we make sure that the t_{std} are less than or equal 5% of t_{avg}, so as to exclude bad hardware and suspicious system states.

Block size selection

We evaluate the performance of block *TVM* algorithms by varying the block size with respect to the cache hierarchy. As for the input tensors, we assume square blocks which have equal length b along all dimensions. Recall that a block *TVM* algorithm relies on the existing *TVM* kernels of Section 4.1 being called for each individual block; the size b thus controls the cache level that we block for. We say that a kernel fits level-L cache if the number of elements it touches is less than or equal to $\alpha z_L/s$, where $0 < \alpha \leq 1$ is the cache saturation coefficient, z_L is the level-L cache size in bytes, and s is the size of a single tensor data element in bytes. The saturation coefficient is such that we obtain the typical cache behavior; setting it too close to 1 usually loses performance, while setting it too close to 0 amounts to benchmarking lower level caches. We find that for best performance, b should be even or a multiple of four, presumably to make optimal use of SIMD instructions, and observe typical throughput is attained at $\alpha = 0.5$. Table 1 summarizes our choices for b in the b_{L1}, b_{L2} and b_{L3} columns such that the *TVM* of a block touching $b^d + b^{d-1} + b$ elements fits L1, L2 and L3 cache, respectively. Note that this parameter is not fine-tuned for better performance.

d	b_{L1}	b_{L2}	b_{L3}
2	44	124	1276
3	12	24	116
4	6	10	34
5	4	6	16
6	3	4	10
7	2	4	7
8	2	3	5
9	2	3	4
10	2	-	4

Table 1: Values of b such that square order-d blocks of b^d elements together with the input vector (b elements) and the output block (b^{d-1} elements) all fit in the L1, L2, and L3 caches, assuming double-precision tensor and vector elements. Note that there is no integer b for $d = 10$ such that the *TVM* routine fits the L2 cache without also fitting L1 cache.

Implementation

The *tvUnfold* (Algorithm 2) relies on a custom memcpy routine *ntmemcpy*, which flags the source memory for early cache eviction. This leads to better performance once the matrix U is multiplied (line 14), since the full cache is available to work on the unfolded tensor. The *ntmemcpy* uses non-temporal reads followed by aligned or unaligned streaming writes, as appropriate.

Both *tvUnfold* and *tvLooped* rely on the *mv* and *vm* matrix–vector multiplications kernels. We always use MKL if these kernels are used on unblocked tensors, but, when called for *MVMs* on individual blocks of a blocked layout, we also consider LIBXSMM [11]—a library especially optimized for repeated dense small matrix–matrix multiplications. We observe that the performance of the *MVM* kernel strongly depends on the ratio between rows and columns: for short-wide and tall-skinny matrices LIBXSMM usually outperforms MKL, while otherwise MKL exhibits better performance. We tune the selection of MKL or LIBXSMM to our Ivy Bridge machine based on the aspect ratio, the kernel orientation (*mv* or *vm*), and the number of bytes the computation touches.

5.2. Microbenchmarks

As *TVM* algorithms are bandwidth-bound, we retrieve an upper bound on their performance by benchmarking the peak bandwidth our machine attains

Copy kernel	L1	L2	L3	RAM
memcpy	88.41	46.15	29.33	7.62
ntmemcpy	83.78	46.25	25.98	11.39

Table 2: Sample effective bandwidths (in GB/s) of the copy kernels when copy size fits into different levels of the memory hierarchy. These are representative values taken from Figure 1 at 16 KB for L1, 128 KB for L2, 16 MB for L3, and 8 GB for RAM.

in practice. We benchmark using STREAM variants, the C standard *memcpy*, and the hand-coded *ntmemcpy*. To measure an upper bound on computation time for in-cache blocks, we separately benchmark the *mv* and *vm* *MVM* kernels for cache-sized matrices and for much larger matrices as well, as a proxy for the overall expected *TVM* performance.

We also investigate the performance and mode-obliviousness of using one of the state-of-the-art *TVM* algorithms on tensors that fit in cache, as this is the inner kernel of the block *TVM* algorithms. Since bandwidth is the overall limiting factor, all measurements are in Gigabyte per second (GB/s).

Upper bounds on effective bandwidth

We measure the maximum bandwidth of our system using several variants of the STREAM benchmark, reporting the maximum measured performance only. Using the full machine we attain 76.7 GB/s (using two processors and ten threads each); however, since our proposed *TVM* algorithms are sequential, STREAM performance of a single core yields the upper bound of interest at 18.3 GB/s. Both results are consistent with the theoretical peak.

The tensor blocked layouts require blocks to be streamed from RAM into cache. For the *tvUnfold*, we unfold the tensor with a series of copies. Figure 1 benchmarks the standard *memcpy* and *ntmemcpy* for different sizes, including the cache-sized copies in order to attain upper bounds for data movements when processing a single block. The *ntmemcpy* performance is indeed better for RAM-sized copies since it avoids caching source memory areas. Table 2 summarizes the results by selecting representative performance figures for each level of the memory hierarchy.

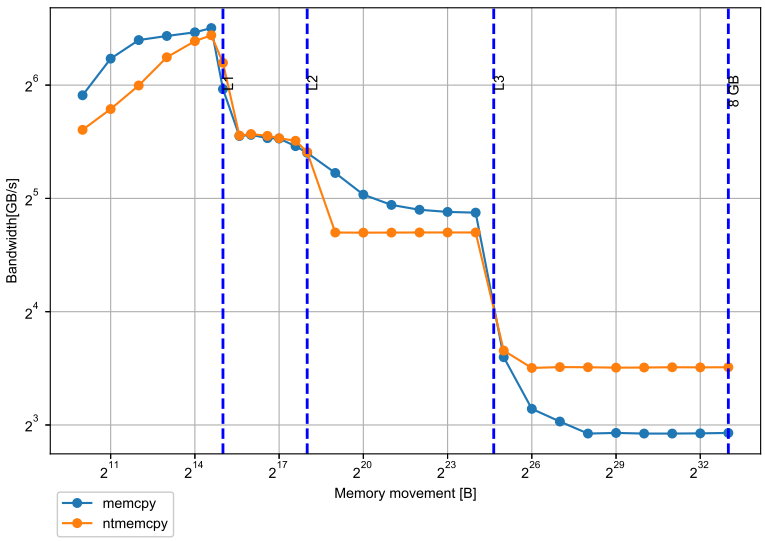

Figure 1: Plot of the effective bandwidth (in GB/s) of the copy kernels versus the amount of bytes moved by the copies, in bytes. The results are stable for sizes larger than 8 GB.

Matrix–vector multiplication using mv and vm

The *mv* or *vm* are the innermost kernels of all *TVM* algorithms: the block algorithms call either *tvLooped* or *tvUnfold* algorithms for individual blocks, while those two algorithms, in turn, execute one or more *MVM* kernels. To gauge the overall computational performance of both in-cache matrices and *tvLooped* and *tvUnfold* algorithms, we benchmark the speed of single calls to *mv* and *vm* over a range of d-dimensional tensors interpreted as tall-skinny or short-wide matrices. Table 3 summarizes the results for matrices that do not fit in cache, while Table 4 contains those for in-cache matrices.

For large matrices, the *mv* has better performance than the *vm*, since the former operates on the output vector via a single stream, while the latter is forced to either i) access the input matrix with stride, or ii) access the output vector multiple times; which both result in reduced performance. The *mv* attains better performance than Table 2 would predict presumably because of the

d	*mv*	*vm*
2	12.79	11.64
3	14.25	9.58
4	10.79	9.83
5	9.45	9.85
6	11.29	9.71
7	12.77	10.03
8	13.52	9.82
9	13.46	10.72
10	13.16	9.43

Table 3: Effective bandwidth (in GB/s) of a single *mv* and *vm*, given a tall-skinny and short-wide $\rho_{(0,1)}$-matrix, respectively. Matrix sizes n are such that at least several GBs of memory are required. The order d determines the aspect ratio of the matrix as n^{d-1} to n. All experiments use MKL.

	mv			*vm*		
d	b_{L1}	b_{L2}	b_{L3}	b_{L1}	b_{L2}	b_{L3}
2	*35.42*	***37.84***	*26.46*	**45.25**	*38.94*	*26.13*
3	*26.21*	***32.51***	*26.18*	38.26	***42.54***	*22.81*
4	20.94	***27.29***	*25.09*	**43.14**	40.07	26.77
5	19.29	*21.08*	***23.39***	29.76	**35.95**	21.26
6	18.47	*18.34*	***23.58***	23.56	**33.70**	31.10
7	12.01	**20.53**	19.84	11.18	**33.36**	28.86
8	14.43	**19.14**	18.96	19.09	**35.88**	27.76
9	16.82	**19.27**	18.59	29.35	**33.27**	26.47
10	18.51	-	18.58	**42.45**	-	26.50

Table 4: Effective bandwidth (in GB/s) of a single *mv* (left) or *vm* (right) given a $b^{d-1} \times b$ and $b \times b^{d-1}$ $\rho_{(0,1)}$-matrix, respectively, with b as defined in Table 1. MKL results are in *italics* while LIBXSMM results are in regular font; we report only the best-performing variant. The best results for any given d are in **bold**. Note that there is no integer b for $d = 10$ such that the *TVM* routine fits the L2 cache without also fitting L1 cache.

reuse of cached input vector elements, which could only benefit the *vm* if it was implemented using accesses with stride.

Comparing the results for cache-sized matrices to Table 2 would indicate that for L1-sized and L2-sized matrices, the *MVM* becomes compute-bound. We also observe that the *vm* outperforms *mv*, especially for lower cache levels and higher d, and that blocking for L2 typically is preferred. Furthermore, the *mv* exhibits slowdowns when the aspect ratio increases while the *vm* is oblivious to it; we exploit this property to attain mode-oblivious behavior for the block *TVM* algorithms.

Single-block tensor–vector multiplication

We discard *tvUnfold* as the inner kernel of the block *TVM* algorithms because it requires a complete unfold of each block which would double the data movement, resulting in a major performance overhead at least for L2- and L3-sized tensors. Therefore, we will only use *tvLooped* as the inner kernel of the block *TVM* algorithms.

We measure the performance of *tvLooped* (Algorithm 1) on cache-sized tensors in Table 5, for each mode $0 \leq k < d$, and report the average performance over all modes (left). Additionally, we measure the unbiased sample standard deviation between the modes (right) as a measure of mode-obliviousness—the lower this value, the more consistent the performance when computing in arbitrary modes. The *tvLooped* algorithm uses the *vm* kernel for all modes $k < d-1$ and uses the slower-performing and less mode-oblivious *mv* only for the mode $k = d - 1$.

We achieve best performance on L2-sized tensors, which is consistent with the microbenchmarks for the *vm* and *mv* kernels. However, we achieve best mode-obliviousness for L3-sized blocks, while a lower d benefits both performance and mode-obliviousness for both L2 and L3. Since in practical application we still require retrieving the input blocks from main memory and since both L2 and L3 performances are higher than the bounds in Table 3, we expect the best results for L3-sized blocks.

	Average performance			Sample stddev.		
d	b_{L1}	b_{L2}	b_{L3}	$\mathrm{std}_{b_{\mathrm{L1}}}$	$\mathrm{std}_{b_{\mathrm{L2}}}$	$\mathrm{std}_{b_{\mathrm{L3}}}$
2	**40.34**	38.39	25.84	17.23	2.03	**0.68**
3	33.08	**35.56**	25.11	18.74	17.05	**7.83**
4	31.95	**33.69**	24.76	35.14	16.29	**5.86**
5	24.30	**29.64**	22.87	21.66	21.03	**17.96**
6	20.11	27.53	**28.37**	18.69	24.24	**11.48**
7	10.31	**28.96**	25.70	**11.98**	19.45	15.60
8	15.57	**26.39**	24.64	20.66	27.39	**14.48**
9	22.64	**26.72**	25.55	29.83	24.29	**13.21**
10	**30.68**	-	24.58	36.81	-	**11.83**

Table 5: Average effective bandwidth (in GB/s) and relative standard deviation of *tvLooped* (Algorithm 1), in percentage versus the average bandwidth over all possible $k \in \{0, 1, \ldots, d-1\}$. The input tensor is a square block of size b as given in Table 1. The highest bandwidth and lowest standard deviation for each d are stated in **bold**. Note that there is no integer b for $d = 10$ such that the *TVM* routine fits the L2 cache without also fitting L1 cache.

5.3. The state-of-the-art tensor–vector multiplication algorithms

This section benchmarks *tvLooped* and *tvUnfold* algorithms for large tensors with general unfolded layouts ρ_π: *tvlooped* (Algorithm 1), which repeatedly makes BLAS2 calls over the given input tensor as-is, and *tvUnfold* (Algorithm 2) which first unfolds the input tensor into a layout appropriate for the multiplication mode and then calls BLAS2 once.

5.3.1. The tvLooped algorithm

Table 6 shows the results of *tvLooped* for $k \in \{1, \ldots, d-2\}$; like for Table 3, these are higher than the raw memory-copy speeds in Table 2 due to cache reuse. We omit the results for $k = 0$ and $k = d-1$ since their equivalence to a single *vm* and *vm*, respectively, for which the results are in Table 3.

We previously learned that mode $k = d-1$ (*mv*) is preferred over $k = 0$ (*vm*) for matrices that cannot be cached; Table 6, however, shows several modes $0 < k < d-1$ that exhibit higher performance than a single *mv* call. This is due to *tvLooped* dividing the computation into multiple *vm* calls on smaller matrices: when the output matrix fits cache, the number of cache misses may be significantly reduced. To test this hypothesis, the results which correspond to calls to *vm* on matrices for which the output matrix fits L2 cache size (and not L1) are printed in *italic*, while marking the best results in **bold**. From the

$d\backslash k$	1	2	3	4	5	6	7	8
3	***13.57***	-	-	-	-	-	-	-
4	***11.28***	11.25	-	-	-	-	-	-
5	11.16	***13.03***	10.05	-	-	-	-	-
6	9.92	10.98	***13.11***	10.87	-	-	-	-
7	10.18	11.32	10.81	***12.81***	10.89	-	-	-
8	10.24	11.44	11.39	***12.54***	10.46	9.62	-	-
9	10.41	10.70	10.64	11.32	***12.56***	9.85	7.72	-
10	9.18	9.86	10.84	9.08	11.67	***12.59***	10.11	6.65

Table 6: Effective bandwidth (in GB/s) of *tvLooped* (Algorithm 1), for $k \in \{1, \ldots, d-2\}$ for each d. Tensor sizes n are such that at least several GBs of memory are required. For each d, the best bandwidth (between modes 1 and $d-2$) is in bold, while the result for which the MVM kernel fits L2 is in italics. All experiments use MKL.

d	avg_{tvL}	std_{tvL}
2	12.22	6.66
3	12.47	20.24
4	10.79	6.27
5	10.71	13.49
6	10.98	11.07
7	11.26	10.07
8	11.13	12.29
9	10.82	14.98
10	10.26	18.56

Table 7: The average effective bandwidth (in GB/s) and relative standard deviation (in percentage versus the average bandwidth) of *tvLooped*, for all possible $k \in \{0, 1, \ldots, d-1\}$ for each d. Tensor sizes n are such that at least several GBs of memory are required. All experiments use MKL.

table, we indeed observe that the fastest results for given d are obtained for computations making optimal use of the L2 cache.

For all d, one may observe that the performance decreases with increasing k, even if the output matrix fits in the L2 cache. This conforms to the data movement overhead (4.3) of input vector elements. Table 7 (left) summarizes the measured speed averaged over all modes, for each d, together with the the standard deviation between the modes (right).

5.3.2. The tvUnfold algorithm

Table 8 benchmarks *tvUnfold* (Algorithm 2) for large tensors for $k \in \{1, \ldots, d-2\}$. The results for modes 0 and $d-1$ are equivalent to those of a single *vm* and *vm* in Table 3, respectively.

$d\backslash k$	1	2	3	4	5	6	7	8	avg_{tvU}	std_{tvU}
2	-	-	-	-	-	-	-		12.22	6.66
3	3.48	-	-	-	-	-	-	-	6.36	83.75
4	3.62	3.46	-	-	-	-	-	-	4.50	80.03
5	3.64	3.63	3.07	-	-	-	-	-	3.76	77.73
6	3.65	3.64	3.65	1.95	-	-	-	-	3.46	88.34
7	3.74	3.73	3.73	3.69	3.46	-	-	-	3.64	81.39
8	3.78	3.77	3.78	3.78	3.71	3.72	-	-	3.68	75.93
9	3.95	3.94	3.94	3.94	3.93	3.79	3.71	-	3.54	77.26
10	3.90	3.90	3.90	3.90	3.87	3.81	3.70	3.24	3.77	74.71

Table 8: Effective bandwidth (in GB/s) of *tvUnfold* (Algorithm 2), for $k \in \{1, \ldots, d-2\}$ for each d. The two columns on the right are the average effective bandwidth (in GB/s) and relative standard deviation (in percentage versus the average bandwidth) for $k \in \{0, 1, \ldots, d-1\}$. Tensor sizes n are such that at least several GBs of memory are required. All experiments use MKL.

The *tvUnfold* performance is suboptimal as it is equivalent to performing two operations in sequence: a large memory copy (the unfold), followed by a single *vm*. These are bounded by the copy and *MVM* microbenchmarks of Section 5.2, respectively. The performance of *tvUnfold* thus is half of the fastest one at best and half of the slowest one at worst; indeed, most results are very close to the raw *memcpy* performance. The effective bandwidths are often $3x$ slower than those achieved by *tvLooped* and performance is highly unpredictable given standard deviations of up to 88% of average performance; unfold-based *TVM* implementations should be avoided.

5.4. Block tensor–vector multiplication algorithms

Here we benchmark the two blocked tensor layouts proposed in Section 4.2: $\rho_\pi\rho_\pi$ and $\rho_Z\rho_\pi$. We expect both block algorithms to improve mode-obliviousness over *tvLooped* and *tvUnfold*, and, for sufficiently small block sizes, expect the $\rho_Z\rho_\pi$-algorithm to cache-obliviously improve reuse of input vector and output tensor elements. Conforming to earlier experiments, we only consider the *tvLooped* (Algorithm 1) for performing the *TVM* on a single block by fixing the *tv* parameter to the block *TVM* (Algorithm 3).

Tables 9 and 10 contain the experimental results of $\rho_\pi\rho_\pi$-block and $\rho_Z\rho_\pi$-block algorithm, respectively. The $\rho_Z\rho_\pi$-block algorithm achieves a mode-oblivious behavior similar to that of *tvLooped*, while both performance and

	Average performance			Sample stddev.		
d	b_{L1}	b_{L2}	b_{L3}	$\mathrm{std}_{b_{\mathrm{L1}}}$	$\mathrm{std}_{b_{\mathrm{L2}}}$	$\mathrm{std}_{b_{\mathrm{L3}}}$
2	9.25	9.92	**13.94**	11.54	2.57	**2.48**
3	8.67	**11.81**	10.29	46.28	11.79	**6.80**
4	6.42	**11.30**	11.12	56.47	15.07	**12.79**
5	5.40	9.62	**10.61**	47.43	29.69	**10.82**
6	3.79	7.71	**11.47**	56.02	27.53	**10.78**
7	3.04	6.58	**8.97**	51.98	**41.09**	45.89
8	3.27	4.67	**8.11**	48.10	49.31	**43.83**
9	3.47	4.99	**7.30**	44.63	46.40	**35.82**
10	3.67	-	**7.77**	42.16	-	**33.79**

Table 9: Average effective bandwidth (in GB/s) and relative standard deviation (in percentage versus the average bandwidth) of the $\rho_\pi\rho_\pi$-block algorithm with *tvLooped*, for $k \in \{0, 1, \ldots, d-1\}$ for each d. Blocked tensor sizes n are such that at least several GBs of memory are required, while block sizes defined in Table 1 hitting different L1, L2 and L3 cache.

	Average performance			Sample stddev.		
d	b_{L1}	b_{L2}	b_{L3}	$\mathrm{std}_{b_{\mathrm{L1}}}$	$\mathrm{std}_{b_{\mathrm{L2}}}$	$\mathrm{std}_{b_{\mathrm{L3}}}$
2	10.43	9.93	**13.91**	5.14	3.21	**2.75**
3	**12.30**	11.67	10.35	6.58	11.99	**6.32**
4	11.73	**12.13**	11.31	**6.19**	7.71	10.23
5	10.89	**11.71**	11.17	**4.14**	6.01	10.06
6	10.03	10.88	**10.99**	10.32	**4.69**	15.08
7	8.70	10.74	**11.03**	13.10	**4.62**	8.40
8	9.10	10.13	**10.87**	11.13	7.96	**5.85**
9	9.25	10.21	**10.34**	8.88	**7.39**	9.44
10	9.55	-	**10.56**	**8.22**	-	9.17

Table 10: Average effective bandwidth (in GB/s) and relative standard deviation (in percentage versus the average bandwidth) of the $\rho_Z\rho_\pi$-block algorithm with *tvLooped*, for $k \in \{0, 1, \ldots, d-1\}$ for each d. Blocked tensor sizes n are such that at least several GBs of memory are required, while block sizes defined in Table 1 hitting different L1, L2 and L3 cache.

mode-obliviousness of the $\rho_\pi\rho_\pi$-block algorithm drop as d increases. Compared to the $\rho_Z\rho_\pi$-block algorithm, performance losses are up to 67% while standard deviations are multiplied several times. This attests that the natural order blocking alone is not enough to induce mode-oblivious behavior; the Morton order based blocking is necessary.

In line with experiments from Section 5.2, both block algorithms generally achieve the highest performance for L3-sized blocks, and if not, in all but two cases achieve it on L2-sized blocks instead. In terms of mode-obliviousness, the $\rho_Z\rho_\pi$-block algorithm performs best using L2- or L3-sized blocks. Blocking for

d	*tvUnfold*	*tvLooped*	taco	$\rho_\pi\rho_\pi$-block	$\rho_Z\rho_\pi$-block	$\rho_Z\rho_\pi^\star$-block
2	12.22	12.22	9.36	13.94	13.91	**14.1**
3	6.36	**12.47**	11.92	10.29	10.35	11.06
4	4.50	10.79	10.09	11.12	11.31	**11.86**
5	3.76	10.71	10.69	10.61	11.17	**12.06**
6	3.46	10.98	9.93	11.47	10.99	**11.48**
7	3.64	11.26	9.55	8.97	11.03	**11.52**
8	3.68	**11.13**	6.94	8.11	10.87	10.87
9	3.54	**10.82**	6.75	7.30	10.34	10.36
10	3.77	10.26	7.05	7.77	10.56	**10.62**

Table 11: Average effective bandwidth (in GB/s) of different algorithms for large order-d tensors. The highest bandwidth, signifying the best performance, for each d is shown in **bold**. Tensor sizes n are such that at least several GBs of memory are required.

L2 incurs a small performance penalty, however, so blocking for L3 is preferred. The $\rho_Z\rho_\pi$-block *TVM* maintains high performance and low standard deviations across all values of d tested; the increase in cache efficiency on input and output elements the Morton order induces proves crucial to blocked tensor layouts. This algorithm performs slower than $\rho_\pi\rho_\pi$ only for $d = 2$ and 6, and only slightly so.

We measure the highest performance for order-2 tensors at almost 14 GB/s for L3-sized blocks, for both block algorithms. This is higher than the raw *vm* and *mv* performance from Section 5.2 and 1.5 GB/s higher than the unblocked *tvLooped*, showing the benefit of a blocked data layouts even for regular matrices.

5.5. Comparisons

This section compares all presented *TVM* algorithms with the code generated by The Tensor Algebra Compiler (taco). Tables 11 and 12 compare the average bandwidths and standard deviations of the state-of-the-art algorithms, the taco-generated *TVM* kernels, the block algorithms $\rho_\pi\rho_\pi$, $\rho_Z\rho_\pi$ and $\rho_Z\rho_\pi^\star$, where the last one uses a hand-tuned blocking parameter. For the blocked layouts, we select the block sizes b_{L3} in Table 1 which correspond to $0.5z_{\mathrm{L3}}$ for reasons discussed in the previous subsection. For the $\rho_Z\rho_\pi^\star$ block algorithm, we used a block size such that the *TVM* of a block fits a memory of size $0.1z_{\mathrm{L3}}$ which not only performs better, but also is more suitable for any future threaded use of the Morton block layout.

While taco enables very generic generation of possibly quite complex tensor

d	*tvUnfold*	*tvLooped*	taco	$\rho_\pi\rho_\pi$-block	$\rho_Z\rho_\pi$-block	$\rho_Z\rho_\pi^\star$-block
2	6.66	6.66	18.50	2.48	2.75	**0.65**
3	83.75	20.24	38.25	6.80	**6.32**	12.99
4	80.03	6.27	38.18	12.79	**10.23**	10.31
5	77.73	13.49	33.65	10.82	10.06	**7.08**
6	88.34	11.07	30.30	10.78	15.08	**8.58**
7	81.39	10.07	28.50	45.89	8.40	**4.73**
8	75.93	12.29	11.53	43.83	5.85	**5.82**
9	77.26	14.98	10.21	35.82	9.44	**9.44**
10	74.71	18.56	11.03	33.79	9.17	**9.17**

Table 12: Relative standard deviation (in percentage versus the average bandwidth) of different algorithms for large order-d tensors. The lowest standard deviation, signifying the best mode-oblivious behavior, for each d is shown in **bold**. Tensor sizes n are such that at least several GBs of memory are required.

computation codes and its performance improves on the *tvUnfold* except for $d = 2$, it lags behind all other variants except once for $\rho_\pi\rho_\pi$ at $d = 7$. The lack of performance is explained by taco not reverting to optimized BLAS2 kernels in its generated codes. It also does not generate mode-oblivious code until $d > 7$, curiously reaching parity with the $\rho_Z\rho_\pi$-blocked results for $d = 9$ and 10. Between the two blocked variants, the $\rho_Z\rho_\pi$-block generally performs better in both performance and mode-obliviousness, while the $\rho_\pi\rho_\pi$ blocked layout additionally incurring unusably high standard deviations for $d > 5$. Both block algorithms dominate *tvUnfold* and taco in terms of performance (except for $d = 7$) and mode-obliviousness. The $\rho_Z\rho_\pi^\star$-block algorithm dominates all other variants except *tvLooped* for $d = 8, 9$, where they perform equally. Considering mode-obliviousness, the block algorithms following the $\rho_Z\rho_\pi$ and $\rho_Z\rho_\pi^\star$ tensor layouts achieve the best results by very comfortable margins.

5.6. Case study: The higher-order power method

Here we study the proposed tensor layouts in the context of the iterative higher-order power method (HOPM) [5, 6]. Given a square d-dimensional tensor and d initial vectors the HOPM proceeds as in Algorithm 6. Each of the $d(d-1)$ *TVM* calls per iteration can be computed using *tvLooped*; this requires a buffer space of n^{d-1} and yields a straightforward baseline implementation. The number of floating point operations is $d(3n + \sum_{i=2}^{d} 2n^i)$ per iteration for

both the normalization and the *TVMs*. The number of data elements touched per iteration is d^2n for all vectors, plus $d(n^d + \sum_{i=2}^{d-1} 2n^i)$ for streaming the input tensor and repeatedly streaming intermediate tensors, plus $2dn$ for the normalization step. Hence the arithmetic intensity is

$$\frac{d\left(3n + \sum_{i=2}^{d} 2n^i\right)}{wd\left(2n + dn + n^d + \sum_{i=2}^{d-1} 2n^i\right)} \textit{ flop per byte}\,, \tag{5.1}$$

with w being the number of bytes required to store a single element. Like for the arithmetic intensity of the *TVM* 4.2, this is bounded from above by $2/w$; the HOPM remains a bandwidth-bound operation.

When assuming a block layout, however, the loops on lines 4–7 can be implemented as a single kernel: the *tensor times a sequence of vectors*, or *ttsv*. Just as the block *TVM* algorithm calls *tvLooped* on each block, our blocked layouts allow making $d-1$ *tvLooped* calls on that single block. Compared to non-blocked layouts, we expect significant gains due to computing each of the d batches of $d-1$ *TVMs* entirely in cache, while for Morton-ordered blocks we additionally expect to observe an accumulated gain from increased cache efficiency on the input and output vectors. The blocked *ttsv* requires a buffer with size bounded by b^d, which coincides perfectly with our choice for $\alpha = 0.5$ from the preceding *TVM* analyses. The output vectors must be reset to zero before each of the d blocked *ttsv* calls, causing a $\Theta(dn)$ overhead per iteration in both time and data movement. However, this is negligible compared to the $d\sum_{i=2}^{d-1} 2n^i$ intermediate tensor elements it saves from being streamed from main memory.

Tables 13 and 14 show experimental results of a single iteration of HOPM on two different compute nodes, Ivy Bridge and Haswell, respectively. We consider only *tvLooped* and the proposed block algorithms, as Table 11 clearly indicated that these algorithms have better performance than others. We choose the input tensor size n to yield close to 8 GB sized data. For the block algorithms, where n is a multiple of b, we choose the latter to result in block sizes that fit L3 cache. Since HOPM computation involves all tensor modes, we do not evaluate mode-obliviousness in this case study. We always compute the speed in GB/s

Algorithm 6 A basic higher-order power method

Input: A square tensor $\mathcal{A}$ of order d and size n,
d vectors $u^{(k)}$ of size n, the maximum number of iterations *maxIters*
Output: d vectors $u^{(k)}$ of size n

for *iters* $= 0$ to *maxIters* $- 1$ **do**
  for $k = 0$ to $d - 1$ **do**
    $\tilde{u}^{(k)} \leftarrow \mathcal{A}$
    for $t = 0$ to $k - 1$ **do**
      $\tilde{u}^{(k)} \leftarrow \tilde{u}^{(k)} \; \overline{\times_t} \; u^{(t)}$
    for $t = k + 1$ to $d - 1$ **do**
      $\tilde{u}^{(k)} \leftarrow \tilde{u}^{(k)} \; \overline{\times_t} \; u^{(t)}$
    $u^{(k)} \leftarrow \frac{\tilde{u}^{(k)}}{\|\tilde{u}^{(k)}\|}$
return $(u^{(0)}, u^{(1)}, \ldots, u^{(d-1)})$

according to the data movement equation (5.1).

On the Ivy Bridge, the performance of *tvLooped* drops below 11 GB/s for d larger than 3, while the proposed block algorithms generally remain steady between 11–13 GB/s. They improve over *tvLooped* by 14.1 to 30.4 per cent. Haswell has 26.2 per cent less bandwidth per core, as measured by the STREAM variants of Section 5.2. Nonetheless, we achieve speedups of up to 15.4 per cent using block algorithms over *tvLooped*. As expected, the blocked variants' performances in HOPM (Table 13) are better than those reported in Table 11—while their improvement over the RAM memory speed from Table 2 can only be due to improved cache reuse. Since the ratio of input tensor elements versus those of the vectors u_i is high and increases with d, the benefits of cache reuse on vectors for Morton ordered curves decline and even out to memory copy speed. Finally, we give an example of an absolute run time. An iteration of HOPM of an order-5 tensor of 8 GB takes 4.5 seconds with *tvLooped* and 3.5 seconds with $\rho_Z\rho_\pi$-block on the Ivy Bridge. A straightforward implementation with nested for-loops, where the inner kernel multiplies the tensor with all input vectors simultaneously and accumulates into the output vector without using BLAS (as in $\hat{u}_i^{(0)} \leftarrow \hat{u}_i^{(0)} + \mathcal{A}_{i,j,k} u_j^{(1)} u_k^{(2)}$ for an order-3 tensor $\mathcal{A}$), takes 7 seconds; resulting in 6.29 GB/s, twice slower than the proposed $\rho_Z\rho_\pi$-block.

d	*tvLooped*	$\rho_\pi\rho_\pi$-block	$\rho_Z\rho_\pi$-block
2	11.10	13.96	**13.98**
3	**13.99**	9.85	9.80
4	9.64	**11.32**	11.29
5	9.83	12.80	**12.82**
6	10.88	**12.65**	12.63
7	10.90	12.47	**12.50**
8	10.82	12.34	**12.35**
9	10.30	11.74	**11.76**
10	9.69	11.42	**11.46**

Table 13: Average effective bandwidth (in GB/s) of different algorithms for HOPM of large order-d tensors on an Intel Ivy Bridge node. The highest bandwidth, signifying the best performance, for each d is shown in **bold**. Tensor sizes n are such that at least several GBs of memory are required.

d	*tvLooped*	$\rho_\pi\rho_\pi$-block	$\rho_Z\rho_\pi$-block
2	10.22	**10.89**	10.73
3	**12.16**	8.59	8.55
4	8.47	8.86	**8.87**
5	8.65	8.77	**8.79**
6	8.97	9.23	**9.25**
7	8.58	**9.40**	**9.40**
8	8.54	**9.40**	**9.40**
9	8.04	**9.28**	**9.28**
10	7.88	**8.89**	**8.89**

Table 14: Average effective bandwidth (in GB/s) of different algorithms for HOPM of large order-d tensors on an Intel Haswell node with two Intel Xeon E5-2690 v3 processors. Each processor has 12 cores sharing 30 MB of L3 cache and the upper bound on bandwidth per core of 14.5 GB/s. The highest bandwidth, signifying the best performance, for each d is shown in **bold**. Tensor sizes n are such that at least several GBs of memory are required.

6. Conclusion

We propose a Morton-ordered blocked layout for tensors that achieves high performance and mode-oblivious computations. Our TVM algorithms based on this layout perform as good or better than several state-of-the-art and highly-optimized BLAS-driven variants. We achieve our goal of mode-obliviousness, while all other variants perform several factors worse in this respect. We characterized the performance and mode-obliviousness using tensor–vector multiplications only, as this is the most bandwidth-bound operation among the other common ones including TMM, TTM, and the Khatri-Rao product. The proposed layouts trivially extend to other operations such as the higher-order power method. They also transfer to other architectures, with significant speedups on both Ivy Bridge and Haswell nodes.

The best-performing non-blocked TVM algorithm, *tvLooped*, performs similar to the blocked $\rho_Z\rho_\pi$ TVM, but, depending on the mode of interest, computation speeds vary from as high as 14.35 GB/s to as low as 6.65 GB/s. Taking the standard deviation of the average behavior over all modes as a measure of mode-obliviousness, our blocked layout instead induces up to 3.2x more stable behavior; this is of particular benefit to use cases where kernels are not applied over each mode successively, especially when it is not known a priori which modes are of interest. Approaches that use an unfold step followed by an optimized BLAS call, although perhaps still standard practice, are inferior in terms of both performance and mode-obliviousness; bandwidth-bound operations such as the TVM incur the worst-case performance degradation of a factor two, while standard deviations magnify up to 88% of average performance. Also in the case of a compute-bound TMM or TTM one could do without copying the entire input tensor once; which is indeed unnecessary when using our proposed block layouts.

We implemented a tensor-times-a-sequence-of-vectors kernel which computes successive $TVMs$ over all modes using our blocked layouts. This kernel is the computational core of the HOPM. In this kernel, the cache effects of blocking

are magnified, resulting in even more pronounced performance gains for blocked layouts over successive *tvLooped* calls. For the HOPM, the blocking itself causes the largest increase in spatial locality, while for the TVM, the spatial locality is mainly induced by the Morton order of the blocks. We note that none of our improvements can be achieved on the level of BLAS libraries since we require a change in data layout.

As future work, similar blocked algorithms should be designed for the TMM or Khatri-Rao products with tall-skinny matrices. In these cases, we remain in the hard-to-optimize bandwidth-bound regime. If, just as with a TVM, the input matrices are skinny enough to fit in cache, the oblivious behavior induced by the Morton order and exploited by our $\rho_Z\rho_\pi$-layout will magnify cache reuse and thus boost performance further. Considering general compute-bound TMM and TTM products, the use of Morton-ordered blocks is orthogonal to traditional BLAS3 optimizations and will trivially boost cache reuse further [18].

Other future work includes the auto-tuning of non-square block sizes; maintaining a square block is restrictive for the number of choices that may fit in a targeted cache level since the block size grows exponentially with d. Preliminary benchmarks with choosing non-square block sizes indeed show that the $\rho_Z\rho_\pi$-layout achieves higher TVM performance while retaining mode-obliviousness; indeed, even square block-size tuning can result in significant gains ($\rho_Z\rho_\pi^\star$ in Table 11). Other parameters that could be considered for auto-tuning include software prefetch distances and SIMD sizes.

The use of the Hilbert curve instead of the Morton order, even though computationally more expensive to use, will likely even further increase the cache reuse of the bandwidth-bound TVM, tall-skinny TMM, and tall-skinny Khatri-Rao products. The proposed tensor layout and processing method is orthogonal to most parallelization strategies; integration into such parallel schemes [16, 3] is another logical step.

References

[1] B. W. Bader, T. G. Kolda, Algorithm 862: MATLAB tensor classes for fast algorithm prototyping, ACM Transactions on Mathematical Software 32 (4) (2006) 635–653.

[2] B. W. Bader, T. G. Kolda, et al., Matlab tensor toolbox version 2.6, `http://www.sandia.gov/~tgkolda/TensorToolbox/`, visited on 01-30-2019 (February 2015).

[3] G. Ballard, N. Knight, K. Rouse, Communication lower bounds for matricized tensor times Khatri-Rao product, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018, pp. 557–567.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, The MIT Press, Cambridge, MA, 2009.

[5] L. De Lathauwer, P. Comon, B. De Moor, J. Vandewalle, Higher-order power method–Application in independent component analysis, in: Proceedings NOLTA'95, Las Vegas, USA, 1995, pp. 91–96.

[6] L. De Lathauwer, B. De Moor, J. Vandewalle, On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors, SIAM Journal on Matrix Analysis and Applications 21 (4) (2000) 1324–1342.

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson, An extended set of FORTRAN basic linear algebra subprograms, ACM Trans. Math. Softw. 14 (1) (1988) 1–17.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson, Algorithm 656: An extended set of basic linear algebra subprograms: Model implementation and test programs, ACM Trans. Math. Softw. 14 (1) (1988) 18–32.

[9] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, in: Foundations of Computer Science, 1999. 40th Annual Symposium on, IEEE, 1999, pp. 285–297.

[10] K. Hayashi, G. Ballard, Y. Jiang, M. J. Tobia, Shared-memory parallelization of MTTKRP for dense tensors, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '18, ACM, New York, NY, USA, 2018, pp. 393–394.

[11] A. Heinecke, G. Henry, M. Hutchinson, H. Pabst, LIBXSMM: Accelerating small matrix multiplications by runtime code generation, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 84:1–84:11.

[12] Intel math kernel library reference manual, visited on 30-01-2019.
URL `https://software.intel.com/en-us/articles/mkl-reference-manual`

[13] O. Kaya, B. Uçar, Parallel Candecomp/Parafac decomposition of sparse tensors using dimension trees, SIAM Journal on Scientific Computing 40 (1) (2018) C99–C130.

[14] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, S. Amarasinghe, The Tensor Algebra Compiler, Proc. ACM Program. Lang. 1 (OOPSLA) (2017) 77:1–77:29.

[15] T. G. Kolda, B. W. Bader, Tensor decompositions and applications, SIAM Review 51 (3) (2009) 455–500.

[16] J. Li, C. Battaglino, I. Perros, J. Sun, R. Vuduc, An input-adaptive and in-place approach to dense tensor-times-matrix multiply, in: High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for, IEEE, 2015, pp. 76:1–76:12.

[17] J. Li, J. Sun, R. Vuduc, HiCOO: Hierarchical storage of sparse tensors, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'18, ACM, New York, NY, USA, 2018, pp. 19:1–19:15.

[18] K. P. Lorton, D. S. Wise, Analyzing block locality in Morton-order and morton-hybrid matrices, SIGARCH Comput. Archit. News 35 (4) (2007) 6–12.

[19] D. Matthews, High-performance tensor contraction without transposition, SIAM Journal on Scientific Computing 40 (1) (2018) C1–C24.

[20] G. M. Morton, A computer oriented geodetic data base and a new technique in file sequencing.

[21] A. H. Phan, P. Tichavský, A. Cichocki, Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations, IEEE Transactions on Signal Processing 61 (19) (2013) 4834–4846.

[22] S. Smith, G. Karypis, Tensor-matrix products with a compressed sparse tensor, in: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, IA3 '15, ACM, New York, NY, USA, 2015, pp. 5:1–5:7.

[23] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, J. Demmel, A massively parallel tensor contraction framework for coupled-cluster computations, Journal of Parallel and Distributed Computing 74 (12) (2014) 3176–3190.

[24] P. Springer, P. Bientinesi, Design of a high-performance Gemm-like tensor–tensor multiplication, ACM Transactions on Mathematical Software 44 (3) (2018) 28:1–28:29.

[25] F. G. Van Zee, R. A. van de Geijn, BLIS: A framework for rapidly instantiating BLAS functionality, ACM Transactions on Mathematical Software 41 (3) (2015) 14:1–14:33.

[26] D. W. Walker, Morton ordering of 2D arrays for efficient access to hierarchical memory, The International Journal of High Performance Computing Applications 32 (1) (2018) 189–203.

[27] R. C. Whaley, J. J. Dongarra, Automatically tuned linear algebra software, in: Supercomputing, 1998. SC98. IEEE/ACM Conference on, IEEE, 1998, pp. 38–38.

[28] Z. Xianyi, W. Qian, Z. Chothia, Openblas, visited on 30-01-2019 (2014). URL http://xianyi.github.io/OpenBLAS

[29] A. N. Yzelman, R. H. Bisseling, A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve, in: M. Günther, A. Bartel, M. Brunk, S. Schöps, M. Striebel (Eds.), Progress in Industrial Mathematics at ECMI 2010, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 627–633.